```
/*****************************************************************************
#   Copyright 1997 Silicon Graphics, Inc.  ALL RIGHTS RESERVED.
#
#   UNPUBLISHED -- Rights reserved under the copyright laws of the United
#   States.   Use of a copyright notice is precautionary only and does not
#   imply publication or disclosure.
#
#   THIS SOFTWARE CONTAINS CONFIDENTIAL AND PROPRIETARY INFORMATION OF
#   SILICON GRAPHICS, INC. ANY DUPLICATION, MODIFICATION, DISTRIBUTION, OR
#   DISCLOSURE IS STRICTLY PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN
#   PERMISSION OF SILICON GRAPHICS, INC.
#****************************************************************************/

#include <stdio.h>
#include <strings.h>
#include <math.h>
#include "ri.h"
#include "ri_state.h"
#include "ri_shader.h"
#include <GL/gl.h>

/* lightshader is a linked list of light source shaders held outside
   the state defined in the renderman interface.  it is initialized
   with the four default light shaders from the renderman interface --
   ambientlight, distantlight, pointlight, and spotlight (which are
   defined immediately below).  */

static Shader *lightshader = NULL;

/* the renderman interface allows an indefinite number of lights to
   be active at any time; the lights may be either area lights or
   point lights.  multiple lights may use the same light shader, so
   the light list is kept separate from the light shader list.  the
   lights are kept in an linked list, and each light keeps its own
   light shader and corresponding parameters.  we must break up the
   lights into groups of a number equal to or less than the hardware
   supported number of lights.  the total light contribution is
   obtained by making multiple passes over the scene for each block
   of lights and accumulating the results.

   to return a handle to the application, we simply cast the pointer
   to a light source structure to RtLightHandle. */

int __light_off = 0;

void __light_enable(void)
{
    __light_off |= 0x2;

    if( getenv("FRAGMENT") ) {
        glEnable(GL_FRAGMENT_LIGHTING_SGIX);
        glLightEnviSGIX(GL_LIGHT_ENV_MODE_SGIX, GL_REPLACE);
    } else {
        glEnable(GL_LIGHTING);
    }
}
```

```
void __light_disable(void)
{
    __light_off &= ~0x2;

    if( getenv("FRAGMENT") ) {
        glDisable(GL_FRAGMENT_LIGHTING_SGIX);
    } else {
        glDisable(GL_LIGHTING);
    }
}

void __material_set(float *a, float *d, float *s, float n)
{
    /* scale up to mimic prman default */
    n *= 8.;

    /* clamp for opengl */
    if( n>128. )
        n = 128;

    glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,a);
    glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,d);
    glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,s);
    glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,n);

    glFragmentMaterialfvSGIX(GL_FRONT_AND_BACK,GL_AMBIENT,a);
    glFragmentMaterialfvSGIX(GL_FRONT_AND_BACK,GL_DIFFUSE,d);
    glFragmentMaterialfvSGIX(GL_FRONT_AND_BACK,GL_SPECULAR,s);
    glFragmentMaterialfSGIX(GL_FRONT_AND_BACK,GL_SHININESS,n);
}

void __light_set(float *a, float *d, float *s, float *p, float att, float cut)
{
    glLightfv(GL_LIGHT0, GL_AMBIENT, a);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, d);
    glLightfv(GL_LIGHT0, GL_SPECULAR, s);
    glLightfv(GL_LIGHT0, GL_POSITION, p);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, att);
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, cut);

    glEnable(GL_LIGHT0);

    glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_AMBIENT, a);
    glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_DIFFUSE, d);
    glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPECULAR, s);
    glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_POSITION, p);
    glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_CONSTANT_ATTENUATION, 0.);
    glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_LINEAR_ATTENUATION, 0.);
    glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_QUADRATIC_ATTENUATION,
att);
    glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPOT_CUTOFF, cut);

    glEnable(GL_FRAGMENT_LIGHT0_SGIX);
}
```

```
void __spotlight_set(float *dir, float falloff, float cutoffdelta)
{
    glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
    glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, falloff);
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF_DELTA_SGIX, cutoffdelta);

    glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPOT_DIRECTION, dir);
    glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPOT_EXPONENT, falloff);
    glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPOT_CUTOFF_DELTA_SGIX,
cutoffdelta);
}

/*ARGSUSED*/
void __rile_blacklight(unsigned char *PC)
{
    float zero[4] = {0., 0., 0., 0.};

    __light_set(zero,zero,zero,zero,0.,180.);
}

void __rile_ambientlight(unsigned char *PC)
{
    RtFloat *d = (RtFloat *)PC;
    float color[4], intensity;
    float zero[4] = {0., 0., 0., 0.};

    intensity = *d++;
    color[0] = *d++;
    color[1] = *d++;
    color[2] = *d++;
    color[3] = 1.;

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    __light_set(color,zero,zero,zero,0.,180.);
}

/*ARGSUSED*/
void *__rilc_ambientlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    RtFloat *t;
    int i, size;

    size = 4*sizeof(RtFloat);
    t = (RtFloat *)__ri_malloc(size);
    if (t == NULL)
        return NULL;

    t[0] = 1.;
    t[1] = 1.;
    t[2] = 1.;
    t[3] = 1.;

    for( i=0; i<n; i++ ) {
```

```
      if( tokens[i]==RI_INTENSITY ) {
          t[0] = *(float *)values[i];
      } else if( tokens[i]==RI_LIGHTCOLOR ) {
          t[1] = ((float *)values[i])[0];
          t[2] = ((float *)values[i])[1];
          t[3] = ((float *)values[i])[2];
      }
    }

    return( (void *)t );
}

/*ARGSUSED*/
void *__riim_ambientlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    float color[4], intensity;
    int i;
    float zero[4] = {0., 0., 0., 0.};

    intensity = 1.;
    color[0] = 1.;
    color[1] = 1.;
    color[2] = 1.;
    color[3] = 1.;

    for( i=0; i<n; i++ ) {
        if( tokens[i]==RI_INTENSITY ) {
            intensity = *(float *)values[i];
        } else if( tokens[i]==RI_LIGHTCOLOR ) {
            color[0] = ((float *)values[i])[0];
            color[1] = ((float *)values[i])[1];
            color[2] = ((float *)values[i])[2];
        }
    }

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    __light_set(color,zero,zero,zero,0.,180.);

    return NULL;
}

void __rile_distantlight(unsigned char *PC)
{
    RtFloat *d = (RtFloat *)PC;
    float lpos[4], color[4];
    float intensity = 1.;
    float from[3] = { 0., 0., 0.};
    float to[3] = { 0., 0., 1.};
    float zero[4] = {0., 0., 0., 0.};

    intensity = *d++;
    color[0] = *d++;
    color[1] = *d++;
```

```
    color[2] = *d++;
    color[3] = 1.;
    from[0] = *d++;
    from[1] = *d++;
    from[2] = *d++;
    to[0] = *d++;
    to[1] = *d++;
    to[2] = *d++;

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    /* negated in renderman */
    lpos[0] = from[0]-to[0];
    lpos[1] = from[1]-to[1];
    lpos[2] = from[2]-to[2];
    lpos[3] = 0.;

    __light_set(zero,color,color,lpos,0.,180.);
}

/*ARGSUSED*/
void *__rilc_distantlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    RtFloat *t;
    int i, size;

    size = 10*sizeof(RtFloat);
    t = (RtFloat *)__ri_malloc(size);
    if (t == NULL)
        return NULL;

    t[0] = 1.;
    t[1] = 1.;
    t[2] = 1.;
    t[3] = 1.;
    t[4] = 0.;
    t[5] = 0.;
    t[6] = 0.;
    t[7] = 0.;
    t[8] = 0.;
    t[9] = 1.;

    for( i=0; i<n; i++ ) {
      if( tokens[i]==RI_INTENSITY ) {
          t[0] = *(float *)values[i];
      } else if( tokens[i]==RI_LIGHTCOLOR ) {
          t[1] = ((float *)values[i])[0];
          t[2] = ((float *)values[i])[1];
          t[3] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_FROM ) {
            t[4] = ((float *)values[i])[0];
            t[5] = ((float *)values[i])[1];
            t[6] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_TO ) {
```

```
            t[7] = ((float *)values[i])[0];
            t[8] = ((float *)values[i])[1];
            t[9] = ((float *)values[i])[2];
        }
    }

    return ( (void *)t );
}

/*ARGSUSED*/
void *__riim_distantlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    float lpos[4], color[4];
    float intensity = 1.;
    float from[3] = { 0., 0., 0.};
    float to[3] = { 0., 0., 1.};
    int i;
    float zero[4] = {0., 0., 0., 0.};

    intensity = 1.;
    color[0] = 1.;
    color[1] = 1.;
    color[2] = 1.;
    color[3] = 1.;
    from[0] = 0.;
    from[1] = 0.;
    from[2] = 0.;
    to[0] = 0.;
    to[1] = 0.;
    to[2] = 1.;

    for( i=0; i<n; i++ ) {
        if( tokens[i]==RI_INTENSITY ) {
            intensity = *(float *)values[i];
        } else if( tokens[i]==RI_LIGHTCOLOR ) {
            color[0] = ((float *)values[i])[0];
            color[1] = ((float *)values[i])[1];
            color[2] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_FROM ) {
            from[0] = ((float *)values[i])[0];
            from[1] = ((float *)values[i])[1];
            from[2] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_TO ) {
            to[0] = ((float *)values[i])[0];
            to[1] = ((float *)values[i])[1];
            to[2] = ((float *)values[i])[2];
        }
    }

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    /* negated in renderman */
    lpos[0] = from[0]-to[0];
    lpos[1] = from[1]-to[1];
```

```
    lpos[2] = from[2]-to[2];
    lpos[3] = 0.;

    __light_set(zero,color,color,lpos,0.,180.);

    return NULL;
}

void __rile_pointlight(unsigned char *PC)
{
    RtFloat *d = (RtFloat *)PC;
    float intensity = 1.;
    float color[4];
    float from[4] = { 0., 0., 0., 1.};
    float zero[4] = {0., 0., 0., 0.};

    intensity = *d++;
    color[0] = *d++;
    color[1] = *d++;
    color[2] = *d++;
    color[3] = 1.;
    from[0] = *d++;
    from[1] = *d++;
    from[2] = *d++;
    from[3] = 1.;

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    __light_set(zero,color,color,from,1.,180.);
}

/*ARGSUSED*/
void *__rilc_pointlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    RtFloat *t;
    int i, size;

    size = 7*sizeof(RtFloat);
    t = (RtFloat *)__ri_malloc(size);
    if (t == NULL)
        return NULL;

    t[0] = 1.;
    t[1] = 1.;
    t[2] = 1.;
    t[3] = 1.;
    t[4] = 0.;
    t[5] = 0.;
    t[6] = 0.;

    for( i=0; i<n; i++ ) {
        if( tokens[i]==RI_INTENSITY ) {
            t[0] = *(float *)values[i];
        } else if( tokens[i]==RI_LIGHTCOLOR ) {
```

```
            t[1] = ((float *)values[i])[0];
            t[2] = ((float *)values[i])[1];
            t[3] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_FROM ) {
            t[4] = ((float *)values[i])[0];
            t[5] = ((float *)values[i])[1];
            t[6] = ((float *)values[i])[2];
        }
    }

    return( (void *)t );
}

/*ARGSUSED*/
void *__riim_pointlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    float color[4];
    float intensity = 1.;
    float from[4] = { 0., 0., 0., 1.};
    int i;
    float zero[4] = {0., 0., 0., 0.};

    intensity = 1.;
    color[0] = 1.;
    color[1] = 1.;
    color[2] = 1.;
    color[3] = 1.;
    from[0] = 0.;
    from[1] = 0.;
    from[2] = 0.;
    from[3] = 1.;

    for( i=0; i<n; i++ ) {
        if( tokens[i]==RI_INTENSITY ) {
            intensity = *(float *)values[i];
        } else if( tokens[i]==RI_LIGHTCOLOR ) {
            color[0] = ((float *)values[i])[0];
            color[1] = ((float *)values[i])[1];
            color[2] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_FROM ) {
            from[0] = ((float *)values[i])[0];
            from[1] = ((float *)values[i])[1];
            from[2] = ((float *)values[i])[2];
        }
    }

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    __light_set(zero,color,color,from,1.,180.);

    return NULL;
}

void __rile_spotlight(unsigned char *PC)
```

```
{
    RtFloat *d = (RtFloat *)PC;
    float lpos[4], color[4];
    float intensity = 1.;
    float coneangle = 30., conedeltaangle = 5.;
    float beamdistribution = 2.;
    float from[4] = { 0., 0., 0., 1.};
    float to[3] = { 0., 0., 1.};
    float zero[4] = {0., 0., 0., 0.};

    intensity = *d++;
    color[0] = *d++;
    color[1] = *d++;
    color[2] = *d++;
    color[3] = 1.;
    from[0] = *d++;
    from[1] = *d++;
    from[2] = *d++;
    to[0] = *d++;
    to[1] = *d++;
    to[2] = *d++;
    coneangle = *d++;
    conedeltaangle = *d++;
    beamdistribution = *d++;

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    lpos[0] = to[0]-from[0];
    lpos[1] = to[1]-from[1];
    lpos[2] = to[2]-from[2];
    lpos[3] = 1.;

    __light_set(zero,color,color,from,1.,coneangle);
    __spotlight_set(lpos,beamdistribution,conedeltaangle);
}

/*ARGSUSED*/
void *__rilc_spotlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    RtFloat *t;
    int i, size;

    size = 13*sizeof(RtFloat);
    t = (RtFloat *)__ri_malloc(size);
    if (t == NULL)
        return NULL;

    t[0] = 1.;
    t[1] = 1.;
    t[2] = 1.;
    t[3] = 1.;
    t[4] = 0.;
    t[5] = 0.;
    t[6] = 0.;
```

```
    t[7] = 0.;
    t[8] = 0.;
    t[9] = 1.;
    t[10] = 30.;
    t[11] = 5.;
    t[12] = 2.;

    for( i=0; i<n; i++ ) {
        if( tokens[i]==RI_INTENSITY ) {
            t[0] = *(float *)values[i];
        } else if( tokens[i]==RI_LIGHTCOLOR ) {
            t[1] = ((float *)values[i])[0];
            t[2] = ((float *)values[i])[1];
            t[3] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_FROM ) {
            t[4] = ((float *)values[i])[0];
            t[5] = ((float *)values[i])[1];
            t[6] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_TO ) {
            t[7] = ((float *)values[i])[0];
            t[8] = ((float *)values[i])[1];
            t[9] = ((float *)values[i])[2];
      } else if( tokens[i]==RI_CONEANGLE ) {
          t[10] = (*(float *)values[i])*180./M_PI;
      } else if( tokens[i]==RI_CONEDELTAANGLE ) {
          t[11] = (*(float *)values[i])*180./M_PI;
      } else if( tokens[i]==RI_BEAMDISTRIBUTION ) {
          t[12] = *(float *)values[i];
      }
    }

    return( (void *)t );
}

/*ARGSUSED*/
void *__riim_spotlight(char *name, RtInt n, RtToken tokens[], RtPointer
values[])
{
    float lpos[4], color[4];
    float intensity = 1.;
    float coneangle = 30., conedeltaangle = 5.;
    float beamdistribution = 2.;
    float from[4] = { 0., 0., 0., 1.};
    float to[3] = { 0., 0., 1.};
    int i;
    float zero[4] = {0., 0., 0., 0.};

    intensity = 1.;
    color[0] = 1.;
    color[1] = 1.;
    color[2] = 1.;
    color[3] = 1.;
    from[0] = 0.;
    from[1] = 0.;
    from[2] = 0.;
    from[3] = 1.;
    to[0] = 0.;
```

```
    to[1] = 0.;
    to[2] = 1.;
    coneangle = 30.;
    conedeltaangle = 5.;
    beamdistribution = 2.;

    for( i=0; i<n; i++ ) {
        if( tokens[i]==RI_INTENSITY ) {
            intensity = *(float *)values[i];
        } else if( tokens[i]==RI_LIGHTCOLOR ) {
            color[0] = ((float *)values[i])[0];
            color[1] = ((float *)values[i])[1];
            color[2] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_FROM ) {
            from[0] = ((float *)values[i])[0];
            from[1] = ((float *)values[i])[1];
            from[2] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_TO ) {
            to[0] = ((float *)values[i])[0];
            to[1] = ((float *)values[i])[1];
            to[2] = ((float *)values[i])[2];
        } else if( tokens[i]==RI_CONEANGLE ) {
            coneangle = (*(float *)values[i])*180./M_PI;
        } else if( tokens[i]==RI_CONEDELTAANGLE ) {
            conedeltaangle = (*(float *)values[i])*180./M_PI;
        } else if( tokens[i]==RI_BEAMDISTRIBUTION ) {
            beamdistribution = *(float *)values[i];
        }
    }

    color[0] *= intensity;
    color[1] *= intensity;
    color[2] *= intensity;

    lpos[0] = to[0]-from[0];
    lpos[1] = to[1]-from[1];
    lpos[2] = to[2]-from[2];
    lpos[3] = 1.;

    __light_set(zero,color,color,from,1.,coneangle);
    __spotlight_set(lpos,beamdistribution,conedeltaangle);

    return NULL;
}

/* XXX easily could be optimized */

void __rile_lightshader(Shader *lite)
{
    /* do lighting in camera space */
    glPushMatrix();
    glLoadIdentity();

    if( !CurAttributes->light[lite->id] ) {
        __light_off |= 0x1;
    } else {
        __light_off &= ~0x1;
```

```
    }

    if( !CurAttributes->light[lite->id] ) {
        /* the light is disabled, so just load a black light */
      __rile_blacklight(NULL);
    } else {
        /* load a GL light at the light's position and draw the scene with
           that.  The Cl component was precomputed and will be factored in
           later. */
        float zero[4] = { 0, 0, 0, 0 };
        float one[4] = { 1, 1, 1, 1 };

        if (lite->position[0] == 0 && lite->position[1] == 0 &&
            lite->position[2] == 0 && lite->position[3] == 0) {
            /* ambient light */
            glLightfv(GL_LIGHT0, GL_AMBIENT, one);
            glLightfv(GL_LIGHT0, GL_DIFFUSE, zero);
            glLightfv(GL_LIGHT0, GL_SPECULAR, zero);
        } else {
            glLightfv(GL_LIGHT0, GL_AMBIENT, zero);
            glLightfv(GL_LIGHT0, GL_DIFFUSE, one);
            glLightfv(GL_LIGHT0, GL_SPECULAR, one);
            glLightfv(GL_LIGHT0, GL_POSITION, lite->position);
        }

        glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1.);
        glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.);
        glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.);
        glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 180.);

        glEnable(GL_LIGHT0);

        if (lite->position[0] == 0 && lite->position[1] == 0 &&
            lite->position[2] == 0 && lite->position[3] == 0) {
            glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_AMBIENT, one);
            glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_DIFFUSE, zero);
            glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPECULAR, zero);
        } else {
            glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_AMBIENT, zero);
            glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_DIFFUSE, one);
            glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPECULAR, one);
            glFragmentLightfvSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_POSITION,
                                  lite->position);
        }
        glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX,
                             GL_CONSTANT_ATTENUATION, 1.);
        glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX,
                             GL_LINEAR_ATTENUATION, 0.);
        glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX,
                             GL_QUADRATIC_ATTENUATION, 0.);
        glFragmentLightfSGIX(GL_FRAGMENT_LIGHT0_SGIX, GL_SPOT_CUTOFF, 180.);

        glEnable(GL_FRAGMENT_LIGHT0_SGIX);
    }

    glPopMatrix();
}
```

```
/* shading and lighting functions:  for now ambient, diffuse, and specular
   are implemented assuming the gl lighting is satisfactory.  eventually
   a different mechanism will be put in place for general light shaders.

               ambient()
               diffuse(n)
               specular(n,v,roughness)
               phong(n,v,size)
               trace(p,r)
*/

Temp *PsTemp;
Temp *LTemp;
float CurLitePos[4];
RtMatrix light2camera;
RtMatrix camera2light;

static void executeLightShader(Shader *lite)
{
    extern void invert_matrix(RtMatrix a, RtMatrix m);
    extern void copy_matrix(RtMatrix a, RtMatrix m);
    DrawOp drop;

    CurLitePos[0] = 0;
    CurLitePos[1] = 0;
    CurLitePos[2] = 0;
    CurLitePos[3] = 0;
    LTemp = lite->L;

    copy_matrix(lite->m, light2camera);
    invert_matrix(light2camera, camera2light);

    /* run the p-code */
    __shader_parse(lite->name,NULL,NULL,lite->args);

    lite->position[0] = CurLitePos[0];
    lite->position[1] = CurLitePos[1];
    lite->position[2] = CurLitePos[2];
    lite->position[3] = CurLitePos[3];

    __reg_store(lite->Cl, rgba_rgba);
```

```
    /* set alpha to zero for next guy to do looping */
    drop.cscale[0] = 1.;
    drop.cscale[1] = 1.;
    drop.cscale[2] = 1.;
    drop.cscale[3] = 0.;

    glColorMask(0,0,0,1);
    drop.op = __ps_flatpoly;
    __fb_load(&drop);
    glColorMask(1,1,1,1);
}

static Shader *CurLights = NULL;
static Shader *CurLight = NULL;

void __lt_run_lightshaders(Node *scene)
{
    /* run each lightshader and initialize their L and Cl temporaries and
       fill in the position fields with the position or direction */

    extern char *__cur_onoff;
    Shader *lite = CurLights;
    Node *node = scene;
    DrawOp drop;

    PsTemp = new_temp();

    /* need to enable stencil and set it to 0x1 to allow for conditionals */
    drop.cscale[0] = 1.;
    drop.cscale[1] = 1.;
    drop.cscale[2] = 1.;
    drop.cscale[3] = 1.;
    drop.op = __ps_flatpoly;

    glColorMask(0,0,0,0);
    glClear( GL_STENCIL_BUFFER_BIT );
    glStencilFunc(GL_ALWAYS, 0x1, 0x1);
    glStencilOp(GL_REPLACE, GL_KEEP, GL_REPLACE);
    glEnable(GL_STENCIL_TEST);

    __fb_load(&drop);

    glStencilFunc(GL_EQUAL, 0x1, 0x1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    glColorMask(1,1,1,1);

    /* load Ps, in camera space */
    drop.lut = __lut[LUT_3DEYE];
    drop.nscale = 128.;
    drop.cscale[0] = 128.;
    drop.cscale[1] = 128.;
    drop.cscale[2] = 128.;
    drop.op = __ps_tex3deye;
    glColorMask(1,1,1,0);
    while (node) {
        __cur_onoff = node->light;
```

```
        drop.dlop = node->dlist->list;
        drop.att = &node->att;
        __fb_load(&drop);

        node = node->next;
    }
    glColorMask(1,1,1,1);
    __reg_store(PsTemp, rgba_rgba);

    while ( lite ) {
        if ( !lite->L )
            lite->L = new_temp();
        if ( !lite->Cl )
            lite->Cl = new_temp();

        /* Now we can run the light shader. It should store Cl and L in
           those fields in its struct as they're computed */
        if( !strcmp(lite->name, "areafield") ) {
            executeLightShader(lite);
        }
        lite = lite->next;
    }

    free_temp(PsTemp);
    PsTemp = NULL;

    /* clear the framebuffer, since otherwise we leave behind light source
       cruft in the pixels that don't have any geometry covering them */
    glClear(GL_COLOR_BUFFER_BIT);

    /* turn off stencil */
    glDisable(GL_STENCIL_TEST);
}

static void apply_light(Shader *lite, DrawOp *drop)
{
    /* XXX must restore current light state before each light */
    __ri_setattributes(drop->att,NULL);

    CurLight = lite;

    if ( !strcmp(lite->name, "areafield") ) {
        extern void multipass_area_light(Shader *lite, DrawOp *drop);
        /* hack for area lights */
        __light_disable();
        multipass_area_light(lite, drop);
    } else {
        DrawOp cldrop;

        __light_enable();
        __rile_lightshader(lite);
        drop->op(drop);
        __light_disable();

        cldrop.cscale[0] = 1;
        cldrop.cscale[1] = 1;
        cldrop.cscale[2] = 1;
```

```
        cldrop.cscale[3] = 1;
        cldrop.op = __ps_texpoly;
        cldrop.temp = lite->Cl;
        cldrop.lut = lite->Cl->id;
        __fb_mul(&cldrop);
    }
}

static void __lt_light(DrawOp *drop)
{
    Shader *lite = CurLights;

    if( !lite ) {
        return;
    }

    apply_light(lite, drop);
    lite = lite->next;

    if( lite ) {
        Temp *result = new_temp();
        DrawOp resultDrop;
        resultDrop.cscale[0] = 1;
        resultDrop.cscale[1] = 1;
        resultDrop.cscale[2] = 1;
        resultDrop.cscale[3] = 1;

        __reg_store(result, rgba_rgba);

        while( lite ) {
            apply_light(lite, drop);

            resultDrop.op = __ps_texpoly;
            resultDrop.temp = result;
            resultDrop.lut = result->id;
            __fb_add(&resultDrop);
            __reg_store(result, rgba_rgba);

            lite = lite->next;
        }

        free_temp(result);
    }

    __light_disable();
}

void __lt_lighting(DrawOp *drop)
{
    drop->op = __ps_geometry;
    __lt_light(drop);
}

void __lt_lightnorm(DrawOp *drop)
{
    glTexParameteri(GL_NORMAL_TEXTURE_2D_SGIX,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
```

```
    glCopyTexImage2DEXT(GL_NORMAL_TEXTURE_2D_SGIX,0,GL_RGBA,0,
                YRes-CurOptions->vres,CurOptions->hres,CurOptions->vres,0);

    drop->op = __ps_projtexeyenorm;
    __lt_light(drop);
}
```

```
void init_lightshaders(void)
{
    extern void *__rilc_areafieldlight(char *name, RtInt n, RtToken tokens[],
                                       RtPointer values[]);
    /*    install_lightshader("ambientlight",__rilc_ambientlight);
    install_lightshader("distantlight",__rilc_distantlight);
    install_lightshader("pointlight",__rilc_pointlight);
    install_lightshader("spotlight",__rilc_spotlight);
    */
    __shader_install("areafield",&lightshader,__rilc_areafieldlight);
}

RtLightHandle RiLightSourceV(char *name,
                        RtInt n, RtToken tokens[], RtPointer values[])
{
    Shader *l, *s;

    s = __shader_lookup(name,&lightshader);
    if( s==NULL ) {
      fprintf(stderr,"unknown light shader %s\n",name);
      return NULL;
    }

    l = (Shader *)malloc(sizeof(Shader));
    l->name = (char *)malloc(strlen(name)+1);
    strcpy(l->name,name);
    l->id = LightNumber++;
```

```
    CurAttributes->light[l->id] = 1;
    l->shader = s->shader;
    l->args = s->shader(name,n,tokens,values);
    l->L = NULL;
    l->Cl = NULL;
    l->next = NULL;

    glMatrixMode(GL_MODELVIEW);
    if (RenderState & STATE_WORLD) {
      /* inside worldbegin */
      glGetFloatv(GL_MODELVIEW_MATRIX, (GLfloat *)l->m);

      glPushMatrix();
      glLoadMatrixf((GLfloat *)CurOptions->worldtocamera);
      glMultMatrixf((GLfloat *)l->m);
      glGetFloatv(GL_MODELVIEW_MATRIX, (GLfloat *)l->m);
      glPopMatrix();
    } else {
      /* before worldbegin */
      glGetFloatv(GL_MODELVIEW_MATRIX, (GLfloat *)l->m);
    }

    /* add light to current list */
    if( CurLights!=NULL ) {
      CurLights->last->next = l;
      CurLights->last = l;
    } else {
      CurLights = l;
      CurLights->last = l;
    }

    return((RtLightHandle)l);
}

RtLightHandle RiLightSource(char *name, ...)
{
    RtLightHandle l;

    READ_PARAMETERLIST(name);

    l = RiLightSourceV(name,n,tokens,values);

    FREE_PARAMETERLIST;

    return( l );
}

/* in the renderman interface, if area light sources are not supported,
   the effect is to be that of a point light source.  therefore, the
   area light source calls are simply wrappers for the light source
   call.  we may support area lights in the future. */

RtLightHandle RiAreaLightSourceV(char *name, RtInt n,
                                 RtToken tokens[], RtPointer values[])
{
    return( RiLightSourceV(name,n,tokens,values) );
}
```

```
RtLightHandle RiAreaLightSource(char *name, ...)
{
    RtLightHandle l;

    READ_PARAMETERLIST(name);

    l = RiAreaLightSourceV(name,n,tokens,values);

    FREE_PARAMETERLIST;

    return( l );
}

void __light_restore(ShaderList *list, int remove)
{
    ShaderList *l;

    while( list ) {
      CurAttributes->light[list->shader->id] =
                  !CurAttributes->light[list->shader->id];
      if( CurLight==list->shader ) {
          __rile_lightshader(list->shader);
      }
      if( remove ) {
          l = list->next;
          free(list);
          list = l;
      } else {
          list = list->next;
      }
    }
}

void __rile_Illuminate(unsigned char *PC)
{
    RtFloat *t = (RtFloat *)PC;
    RtLightHandle *h = (RtLightHandle *)t;
    Shader *l = (Shader *)(*h);
    int onoff = (int)(t[1]);
    ShaderList *n;

    if( l!=CurLight ) {
      return;
    }

    if( onoff==RI_TRUE ) {
      if( CurAttributes->light[l->id] ) {
          return;
      }
      CurAttributes->light[l->id] = 1;
    } else {
      if( !CurAttributes->light[l->id] ) {
          return;
      }
      CurAttributes->light[l->id] = 0;
    }
```

```
    __rile_lightshader(l);

    n = (ShaderList *)malloc(sizeof(ShaderList));
    n->shader = l;
    n->next = NULL;
    if( CurAttributes->illuminate==NULL ) {
      CurAttributes->illuminate = n;
      CurAttributes->illuminate->last = n;
    } else {
      CurAttributes->illuminate->last->next = n;
      CurAttributes->illuminate->last = n;
    }
}

void __rilc_Illuminate(RtLightHandle light, RtBoolean onoff)
{
    Shader *l = (Shader *)light;
    RtFloat *t;
    RtLightHandle *h;
    ShaderList *n;

    if( onoff==RI_TRUE ) {
      if( CurAttributes->light[l->id] ) {
          return;
      }
      CurAttributes->light[l->id] = 1;
    } else {
      if( !CurAttributes->light[l->id] ) {
          return;
      }
      CurAttributes->light[l->id] = 0;
    }


    t = (RtFloat *)dlist_append(CurDlist,__rile_Illuminate,2*sizeof(RtFloat));
    if (t == NULL)
        return;

    h = (RtLightHandle *)t;
    *h = light;
    t[1] = onoff;

    n = (ShaderList *)malloc(sizeof(ShaderList));
    n->shader = (Shader *)light;
    n->next = NULL;
    if( CurAttributes->illuminate==NULL ) {
        CurAttributes->illuminate = n;
        CurAttributes->illuminate->last = n;
    } else {
        CurAttributes->illuminate->last->next = n;
        CurAttributes->illuminate->last = n;
    }
}

void __riim_Illuminate(RtLightHandle light, RtBoolean onoff)
{
```

```
    Shader *l = (Shader *)light;

    if( onoff==RI_TRUE ) {
      if( CurAttributes->light[l->id] ) {
          return;
      }
      CurAttributes->light[l->id] = 1;
    } else {
      if( !CurAttributes->light[l->id] ) {
          return;
      }
      CurAttributes->light[l->id] = 0;
    }

}

RtVoid RiIlluminate(RtLightHandle light, RtBoolean onoff)
{
    JumpCur->Illuminate(light,onoff);
}

/* any lights that have been created within a worldbegin-end block are
   deleted on the worldend call.  likewise for those lights defined
   within a framebegin-end block.  because we can have at most one
   world block active within one frame block (no multiple nesting),
   the lights will be defined in sequential blocks:  outside frame->
   outside world->inside world.  the world lights will necessarily be
   deleted before the frame lights.  this function deletes all lights
   in a linked list starting with the light passed in. */

void delete_lights(Shader *l)
{
    Shader *s;

    while( l!=NULL ) {
      RiIlluminate((RtLightHandle)l,RI_FALSE);
      s = l->next;
      free(l->name);
        if (l->L != NULL)
            free_temp(l->L);
        if (l->Cl != NULL)
            free_temp(l->Cl);
      free(l);
      l = s;
    }
}

/* when the attributes are popped, we must return the lights to an
   on-off position that they held before the attributes were
   pushed.  here we loop over all lights; we could optimize this
   function pretty easily. */

/*ARGSUSED*/
void set_lights(RtInt v)
{
#if 0
    while( l!=NULL ) {
```

```
        if( v&(1<<l->id) ) {
            RiIlluminate((RtLightHandle)l,RI_TRUE);
        } else {
            RiIlluminate((RtLightHandle)l,RI_FALSE);
        }
        l = l->next;
    }
#endif
}

static Shader *curLight;

static void load_light_variable(Temp *temp)
{
    DrawOp drop;

    drop.cscale[0] = 1.;
    drop.cscale[1] = 1.;
    drop.cscale[2] = 1.;
    drop.cscale[3] = 1.;
    drop.op = __ps_texpoly;
    drop.temp = temp;
    drop.lut = drop.temp->id;
    __fb_load(&drop);
}

int __lt_start_illuminance(Temp *L, Temp *Cl)
{
    curLight = CurLights;

    if( !curLight ) {
        return 0;
    } else {
        load_light_variable(curLight->L);
        __reg_store(L, rgba_rgba);
        load_light_variable(curLight->Cl);
        __reg_store(Cl, rgba_rgba);
        return 1;
    }
}

int __lt_next_light(Temp *L, Temp *Cl)
{
    curLight = curLight->next;

    if( !curLight ) {
        return 0;
    } else {
        load_light_variable(curLight->L);
        __reg_store(L, rgba_rgba);
        load_light_variable(curLight->Cl);
        __reg_store(Cl, rgba_rgba);
        return 1;
    }
}
```